UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

(1)

AD-A196 699

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFIT/CI/NR 88-96

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

TITLE (and Subtitle): IOGEN: TOWARD AN AUTOMATED TOOL FOR PRODUCTION OF RELIABLE AND VALID TEST SUITES.

5. TYPE OF REPORT & PERIOD COVERED: MS THESIS

6. PERFORMING ORG. REPORT NUMBER

AUTHOR(s): MARK ALLAN NORMAN

8. CONTRACT OR GRANT NUMBER(s)

PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: ARIZONA STATE UNIVERSITY

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

CONTROLLING OFFICE NAME AND ADDRESS

12. REPORT DATE: 1988

13. NUMBER OF PAGES: 71

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): AFIT/NR, Wright-Patterson AFB OH 45433-6583

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE

DTIC ELECTE S AUG 0 3 1988 D

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

SAME AS REPORT

18. SUPPLEMENTARY NOTES

Approved for Public Release: IAW AFR 190-1
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583
20 July 88

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract

This thesis addresses enhancements to a technique for generating test cases and modifications to an automated system implementing the technique. This system, IOGen, generates input/output pairs for the Common APSE Interface Set (CAIS) and for Ada programs in general. Ada language topics for which symbolic execution and IOGen do not address are discussed. The detailed design for an enhanced IOGen system is presented. A case study shows that modifications to IOGen enhance its error detection capability. Finally, current and future areas of research for IOGen are presented.

IOGEN : TOWARD AN AUTOMATED TOOL FOR PRODUCTION OF RELIABLE AND VALID TEST SUITES

by

Mark Allan Norman

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

May 1988

IOGEN : TOWARD AN AUTOMATED TOOL FOR PRODUCTION

OF RELIABLE AND VALID TEST SUITES

by

Mark Allan Norman

has been approved

May 1988

APPROVED:

Timothy E. Lindquist, Chairperson

Paul C. Jorgensen

James Collofello / Kathleen Fisher

Supervisory Committee

ACCEPTED:

Department Chairperson

Dean, Graduate College

Abstract

This thesis addresses enhancements to a technique for generating test cases and modifications to an automated system implementing the technique. This system, IOGen, generates input/output pairs for the Common APSE Interface Set (CAIS) and for Ada programs in general. Ada language topics for which symbolic execution and IOGen do not address are discussed. The detailed design for an enhanced IOGen system is presented. A case study shows that modifications to IOGen enhance its error detection capability. Finally, current and future areas of research for IOGen are presented.

Dedication

To my wife, Vonnie,

and my son, Brian.

Acknowledgement

I would like to thank Dr. Timothy Lindquist for his guidance, enthusiasm, and support in the preparation of this thesis. I extend my thanks and appreciation to Dr. Kathleen Mutch and Dr. Paul Jorgensen for their participation as committee members. Last, but not least, I would like to thank Vickey Wood for her hours of explanation and assistance in educating me on ALEX and AYACC.

Table of Contents

List of Figures

Figure

Chapter 1

Introduction

The United States Department of Defense (DoD) has sponsored the development of the Ada programming language. In addition, the DoD has also developed requirements for programming environments called the Ada Programming Support Environment (APSE). The APSE environment provides a set of tools to support all aspects of the Ada software lifecycle. A number of low-level package interfaces to the underlying machine resources is the Common APSE Interface Set (CAIS, pronounced as case) [1]. The primary motivation for the development of the CAIS is environment tool and data transportability. Currently, research is being conducted at Arizona State University to produce an Operational Definition of the CAIS (CAISOD) [11].

The APSE Evaluation and Validation (E&V) Team was formed by the Ada Joint Program Office to initiate the development of technology for validating the conformance of APSE's to relevant standards and to evaluate the performance of APSE components. Adopting CAIS as the basis for APSE's implies other supporting needs which include a CAIS standard, conformance policy, validation capability, and evaluation capability. Currently, work is being conducted to develop a CAIS Implementation Validation Capability (CIVC) for the Standard CAIS.

The CAISOD will be used to aid in the construction of the CIVC. Together with the CAIS specification, the CAISOD will provide a software basis for creating and testing the validation set. The source code for the CAISOD can be analyzed using static testing techniques to identify necessary validation tests. IOGen [8] was created to perform this static analysis using symbolic execution [7]. Jenkins and Lindquist [9] also describe how IOGen can be used to generate tests in a structured approach to program testing.

IOGen was designed by Jenkins as her master's thesis at Arizona State University [8]. IOGen performs a static analysis of an Ada source routine and develops a set of input/output pairs which represent all of the execution paths through the routine. These I/O pairs are determined through a symbolic execution of the routine. After removing implementation dependencies from these pairs, they could then be used to validate/test (in a black box fashion) different implementations of the CAIS (and eventually the CAISOD).

IOGen is composed of three parts: a scanner, a parser, and an I/O pair generator. The scanner recognizes legal Ada tokens from a stream of characters. It assumes that the input stream of characters originate from an Ada source routine which has successfully compiled with no syntax

errors. The parser analyzes the tokens and builds a symbolic execution tree. The I/O pair generator traverses the symbolic execution tree and generates the I/O pairs.

The scanner removes extraneous blanks and comments and produces a set of legal Ada tokens. The original IOGen scanner would not recognize all token types of the language. Current research work on IOGen will replace the scanner by ALEX [10] so that the user need not modify the source code.

The parser is the major component of IOGen. The original IOGen parser is a one pass, top-down, left-to-right parser. An LL(1) grammar (a subset of Ada syntax) is all that this parser could accept. This parser is also being replaced by AYACC [10] (a parser generator that accepts an LR(1) grammar) so that IOGen will be able to accept the entire Ada syntax. The output from the parser is the symbolic execution tree which represents all of the execution paths through the source routine.

The I/O pair generator traverses the symbolic execution tree and produces one pair for each terminal node in the tree. For each I/O pair, the input that caused the particular execution path is matched with the resulting modifications along the path to form the pair. These I/O pairs are the output from the IOGen system. Any information that is dependent on the implementation of the Ada environment is removed from the I/O pairs resulting in

validation/test suites for the source routine. These validation/test suites could then be executed by the Ada source routine.

This thesis discusses some extensions to symbolic execution with respect to the Ada programming language. Chapter 2 covers a general overview of symbolic execution. In Chapter 3, the extensions for symbolic execution with respect to several Ada expressions and constructs are presented. Each extension involves the manner in which a program statement or expression is represented by symbolic execution. Theoretical and practical methods for the symbolic execution of each extension are discussed and analyzed. The detailed design of a symbolic execution tree package for IOGen is the topic of Chapter 4. A case study in presented in Chapter 5 illustrating the enhanced error detection capability the design of Chapter 4 affords. Other areas of research involved with IOGen are briefly discussed in Chapter 6. Finally, Chapter 7 provides some conclusions about the thesis.

Chapter 2

Symbolic Execution and IOGen

One of the major goals of the software development process is verifying that the end product, the computer program, behaves according to its specifications. Symbolic execution is one method of program verification and is the theoretical basis upon which the IOGen system was originally built.

Hantler and King [7] provide a method of specifying the correct behavior of a program by use of input/output assertions and describe one method for showing that the program is correct with respect to those assertions. An input assertion is represented as an ASSUME statement and is inserted at the beginning of a routine. This ASSUME statement places constraints on all inputs for the routine. An output assertion is represented as a PROVE statement and is inserted immediately before the return from a routine. This PROVE statement represents the expected relation between the inputs and outputs. A routine is said to be correct (with respect to its input and output assertions) if the truth of its input assertion upon routine entry insures the truth of its output assertion upon the routine's exit.

In a proof of correctness for a program it is necessary to verify the program correct over all possible inputs. Hantler and King [7] propose using symbolic values to

represent arbitrary program units. By doing this, variables take on symbolic values of their particular type. The symbolic values can be represented as an elementary symbolic variable or expression; an arbitrary string chosen to represent a variable, or an expression in numbers and arithmetic operators. In this thesis, each symbolic value is represented as a single lower case letter.

When a routine is symbolically executed, the input parameters are assigned a symbolic value upon routine entry. As the symbolic execution continues, each occurrence of each variable is replaced by its symbolic value. In the case of assignment statements, only the variables in the right hand side are replaced by their symbolic values. The left hand side then is assigned the resulting symbolic value of the right hand side. The symbolic execution of routines that contain iterative and conditional constructs results in a symbolic execution tree which contains branches. Each path through the tree represents an execution path through the routine. Attached to each path is a predicate, called a path condition (pc), which describes the conditions that cause the path to be executed. The pc is initially given a value determined from the input assertion of the ASSUME statement at the beginning of the symbolic execution. As branches are encountered in the symbolic execution tree, the pc is modified to reflect the particular conditions causing

each path by use of an AND operation. The remainder of this chapter describes input/output (I/O) pair generation using symbolic execution for several Ada constructs.

2.1 I/O Pairs Generation

The CAISOD serves as the input for symbolic execution [8]. Symbolic execution is used by the IOGen system as a means of generating I/O pairs rather than as a means of proving a program correct. A few modifications to Hantler and King's method are incorporated into IOGen in order to facilitate the generation of I/O pairs. One modification is the removal of the ASSUME statement as the input assertion is always true at the beginning of a routine. Also, since the CAISOD has been validated and can be assumed to be correct, the PROVE statement has been removed.

The execution tree generated is used to develop a set of I/O pairs. The input portion of each I/O pair is determined from the path conditions at the bottom of each execution path (or "leaf") in the execution tree. The path condition describes a set of constraints on the inputs of the program which cause a particular path to be executed. Using these constraints, a set of test data can be generated for each execution path through the program. This test data may be used to establish initial values for global variables or input parameters.

The output portion of each I/O pair is generated

according to actions taken along its respective path in the execution tree. The actions taken represent modifications of global variables and output parameters. The following sections discuss symbolic execution for several Ada programming constructs.

2.2 Assignment Statements

Assignment statements are executed by extracting the values assigned to each of the variables in the right hand side of the expression, evaluating the right hand side to reach a particular value, and finally, assigning this value to the variable in the left hand side of the expression. Symbolic execution replaces the variables in the right hand side with symbolic values. Since this resulting expression contains symbols (not actual values) it is left unchanged. This resulting expression then becomes the new value for the left hand side variable.

2.3 If_Then_Else Statements

If_then_else statements are one form of conditional branching statements. Symbolic execution of if_then_else statements begins by replacing all variables in the boolean expression with their symbolic values. Two boolean expressions are formed using the symbolic boolean expression. One represents the true boolean condition. The other represents the false (negated) boolean condition.

The if_then_else statement causes a two-way branch in

the symbolic execution tree. One branch represents the path taken when the boolean expression evaluates to true. This becomes the 'then' path. The other branch represents the path taken when the boolean expression evaluates to false. This becomes the 'else' path. Eventhough the actual execution paths may rejoin at some later point in the program, they will never rejoin in the symbolic execution tree.

The path conditions for the two paths are formed by ANDing the current pc with each of the two boolean expressions. The 'then' path's pc becomes the current pc ANDed with the symbolic boolean expression. Symbolic execution continues with the 'then' statements. The 'else' path's pc becomes the current pc ANDed with the negation of the symbolic boolean expression. Symbolic execution continues with the 'else' statements. If the statement does not contain an else clause, negation of the symbolic boolean

```
function IS_POSITIVE (num : in integer) return boolean is
[1] begin
[2]   if (num >= 0) then
[3]     return true;
[4]   else
[5]     return false;
[6]   end if;
[7] end IS_POSITIVE;
```

FIGURE 1. Function IS_POSITIVE

expression is not necessary. Symbolic execution for both paths continues with the instructions following the if_then_else statement.

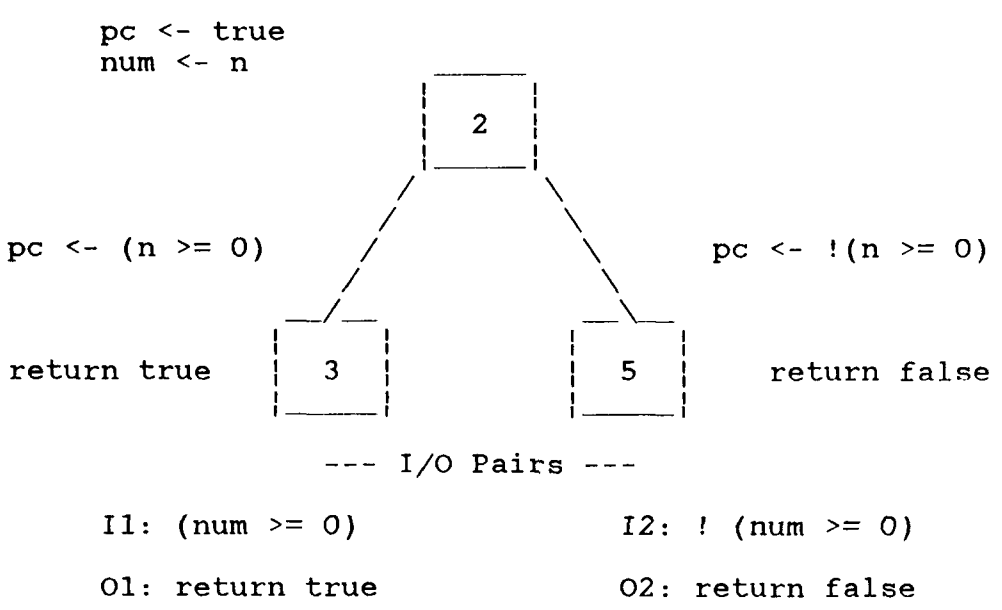


FIGURE 2. Symbolic execution tree: Function IS_POSITIVE

Figure 1 presents an example of a function which returns a boolean value depending on whether the input integer is positive. The symbolic execution tree for this function is shown in Figure 2. The symbolic value 'n' represents the variable num. Figure 2 presents an example of symbolic execution for the if_then_else statement and highlights the separation of paths with their corresponding path conditions.

2.4 Looping Constructs

When a loop is encountered in a routine, it is possible for the symbolic execution tree to be infinite [12]. It is obvious that routines which contain non-terminating loops have infinite execution trees. Furthermore, even when the loops do terminate, the symbolic execution tree may become unmanageably large. Substituting symbolic values for the actual variables during symbolic execution introduces another problem when encountering looping constructs as a unique symbolic value must be generated for each actual variable.

Hantler and King [7] solve this problem by using a form of induction. At the beginning of a loop an inductive assertion is inserted. Each loop is symbolically executed once. This symbolic execution acts as if the loop is a routine all by itself. The path condition upon entrance to the loop is assumed to be true. After symbolically executing the loop once, if the path condition is still true based upon the inductive assertion, then the loop is assumed to be correct for any number of iterations of the loop. In this way, the problems of infinite symbolic execution trees and unmanageably large trees are eliminated.

IOGen takes a similar approach in dealing with looping constructs. I/O pairs are generated for looping constructs based upon only one symbolic execution pass through a loop.

This may not be sufficient for the generation of I/O pairs for programs that contain looping constructs. Jenkins warns that additional tests may be necessary for programs containing looping constructs [8].

The syntactic form of the nontrivial Ada looping constructs involve a boolean condition. The trivial loop is one in which there is no boolean expression and behaves as a simple sequence of statements. The statements of the trivial loop are symbolically executed as if there was no loop.

As with symbolic execution of the if_then_else statement, a two-way branching occurs in the symbolic execution tree for nontrivial loops. One branch represents a single iteration of the loop. The pc at the beginning of the loop is formed by ANDing the current pc with a true loop boolean condition. The other branch represents the execution path around the loop. The pc for this branch is formed by ANDing the current pc with a negated loop boolean condition. Once the symbolic execution for the loop has completed one pass, the pc at that point is ANDed with the false loop boolean condition and the symbolic execution for this path continues with the statement following the loop.

Figure 3 presents a function with a looping construct. This function, TOTAL_POSITIVES, counts the positive integers in an array and returns this value. This function calls the

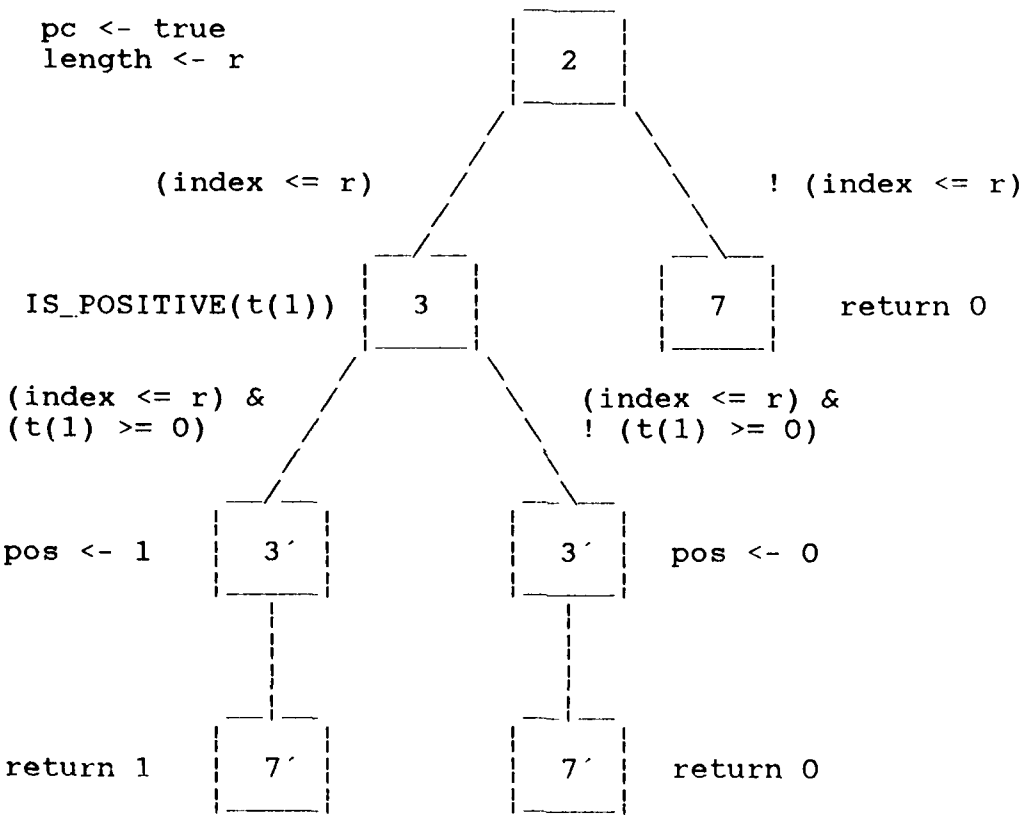
function IS_POSITIVE which is presented in Figure 1. The symbolic execution tree for this function is shown in Figure 4.

```
type table_type is array <> of integer;
index : integer;

function TOTAL_POSITIVES (table : in tabletype;
                          length : integer) return integer is

total : integer := 0;
pos : boolean;

[1] begin

[2]   while index <= length loop
[3]     pos := IS_POSITIVE(table(index));
[4]     total := total + pos;
[5]     index := index + 1;
[6]   end loop;

[7]   return total;
[8] end TOTAL_POSITIVES;
```

FIGURE 3. Function TOTAL_POSITIVES

The symbolic execution tree in figure 4 is an example for the while loop. Symbolic execution for the for loop is identical to the while loop. One branch in the for loop tree represents one iteration of the loop followed by an out-of-range index on the loop. The other branch simply represents an out-of-range index on the loop. The exit_when construct is handled a little different from the while and for loops. One branch from the exit_when represents termination of the loop and execution continues after the loop. The other branch is taken when the loop is not being

exited and execution continues with the next statement in the loop. Again, once the loop has been iterated once, symbolic execution continues with the statements following the loop.



--- I/O Pairs ---

I1: (index<=length)&
(table(1)>=0)
O1: return 1

I2: (index<=length)&
!(table(1)>=0)
O2: return 0

I3: !(index<=length)
O3: return 0

FIGURE 4. Symbolic execution tree: Function TOTAL_POSITIVES

2.5 Case Statements

Symbolic execution for the case statement is very similar to symbolic execution for the if_then_else statement. The difference is that the if_then_else statement provides a two-way branch, while the case statement provides an N-way branch in the symbolic execution tree. The pc for each branch (except for the others choice) is formed by ANDing the current pc with the expression case_selector = 'choice'. The pc for the others choice is formed by ANDing the current pc with the negation of every other path choice. Execution for each branch follows the statements in the respective choices and then continues with the statements following the case statement. Figure 5 shows a sample case statement and Figure 6 presents its symbolic execution tree.

```
[1]  case NUM is
[2]    when 0  =>  temp := 0;
[3]    when 1  =>  temp := 1;
[4]    when others  =>  temp := -1;
[5]  end case;
```

FIGURE 5. Case Statement

2.6 Procedure Calls

There are two general approaches to handling the symbolic execution tree generation for procedure or function calls. One approach is to act as if the procedure body was placed in-line with the code of the calling routine. This

involves a macro-like expansion into the symbolic execution tree [5]. One potential problem with this approach is that care must be taken not to confuse variables local to the procedure with variables i. the calling routine. Also, this approach requires symbolically executing the procedure or function each time it is called.

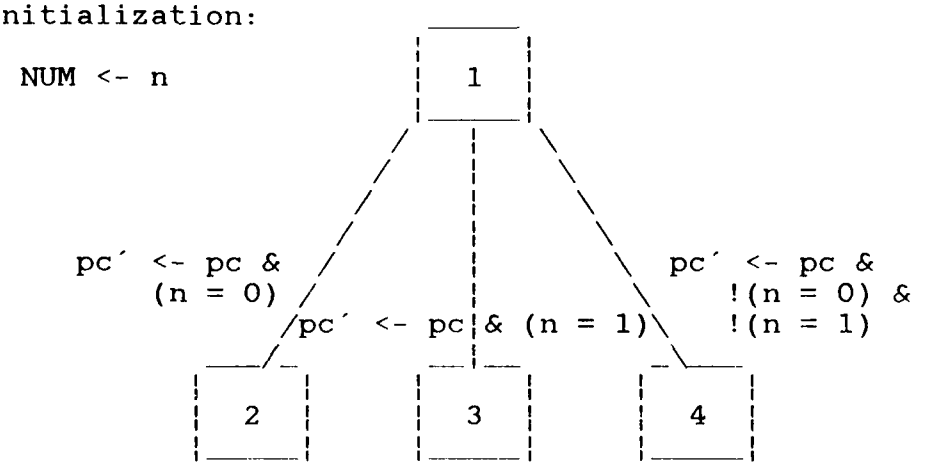


FIGURE 6. Symbolic execution tree: Case Statement

The other approach designed by [5] recommends that the symbolic execution tree be built in a bottom-up fashion. The called procedure or function is symbolically executed and the I/O pairs for that procedure are generated. At the point where the procedure is called in the main routine, an

N-way branch is constructed corresponding to the N I/O pairs from the procedure. The pc for each branch is constructed by ANDing the current pc with the pc input part of each I/O pair from the procedure. This new pc for each branch is expressed in terms of the symbolic values corresponding to the actual parameters. So, the formal parameters are substituted by their corresponding symbolic value.

Figure 4 presents the symbolic execution tree for the function TOTAL_POSITIVES. Figure 2 presents the I/O pairs necessary for the call to function IS_POSITIVE. The two I/O pairs generated cause a two-way branch in the symbolic execution tree of TOTAL_POSITIVES at the point of the function call. Correspondingly, the pc's for each branch are formed by ANDing the current pc with the pc of each branch. The I/O pairs generated from the call to function IS_POSITIVE are:

I1: (index <= r) & (t(1) >= 0)

I2: (index <= r) & !(t(1) >= 0)

2.7 Summary

This chapter has presented an overview of symbolic execution as described by Hantler and King [7] and as designed into the IOGen system by Jenkins [8]. This method of generating I/O pairs has been applied to several Ada constructs to illustrate symbolic execution as implemented through IOGen.

Chapter 3

Ada Language Extensions For Symbolic Execution

Symbolic execution, as described by Hantler and King [7], provides for path coverage testing of a source program. The IOGen system developed by Jenkins [8] is based upon Hantler and King's research. Consequently, IOGen is a tool which assists in the path coverage testing of an Ada source program.

The primary goal of software testing is to provide as much assurance as possible that a program behaves in accordance with its specification. While path coverage testing provides some assurance that a program is correct, it is still possible for errors to remain undetected. A much stronger level of testing is multiple condition coverage testing [4]. Multiple condition coverage testing not only encompasses path coverage testing, it also provides coverage for all the possible input combinations that can cause the various paths to be executed. A program that passes a multiple condition coverage test has a higher level of assurance of its correctness than if it only passes a path coverage test.

The next logical step to take with IOGen is to enhance it so that it provides a set of test cases that constitute a multiple condition coverage test. This is the primary motivation for this thesis. The purpose of this chapter is

two-fold. First, it provides a basis upon which IOGen can be enhanced into a multiple condition coverage testing tool for the Ada language. Second, it extends the theory of symbolic execution for the Ada language in general. These extensions will make symbolic execution a more powerful method of assuring the correctness of Ada programs.

3.1 Overview

This chapter is divided into ten sections based upon expressions, statement types, and Ada constructs that require extra consideration. The categories are: boolean expressions, arithmetic expressions, arrays, attributes, access types, undefined variables, input, discriminants, subtypes, and variant records. One assumption of the IOGen system is that the Ada source program must compile with no errors. This same assumption holds throughout this thesis. The symbolic execution extensions described in this chapter apply only to the run time environment of the program and not to any syntax or type checking errors that may exist.

3.2 Boolean Expressions

Boolean expressions occur in several statement types in order to provide a program the ability to make decisions and act accordingly. Boolean expressions are composed of boolean variables and simple relational expressions by connecting them with logical operators. Boolean variables can take on only the values true and false. Simple

relational expressions compare two expressions to each other or a variable to a value. An example of a simple relational expression is (num >= 5). Simple relational expressions also evaluate to either true or false. In this example, if num is 5 or greater then the expression is true. Otherwise, the expression is false. By connecting boolean variables and relational expressions, rather complex boolean expressions can be constructed.

The logical operators in the Ada language are: AND, OR, XOR, AND THEN, and OR ELSE. The AND operator evaluates to true only when the boolean values on both sides of it are true. The OR operator evaluates to false only when both boolean values are false. The XOR operator evaluates to true only when exactly one of the boolean values is true. The AND THEN operator performs the AND operation only if the left boolean value is true. Otherwise, it evaluates to false. Similarly, the OR ELSE operator performs the OR operation only when the left boolean value is false. Otherwise, it evaluates to true.

For a boolean expression that contains only one boolean variable or simple relational expression, there are only two possible input combinations for the expression. These, of course, are true and false. However, if the boolean expression contains more than one boolean variable or simple relational expression, then there are more than

two possible input combinations to the expression. Since each boolean variable and simple relational expression has two possible values, the number of input conditions on a boolean expression is a power of two. This power is just the number of individual boolean variables and simple relational expressions in the boolean expression. Figure 7 shows an expamle of a boolean expression with three boolean variables A, B, and C. The table shows that there are eight possible input combinations. This agrees with the formula of two raised to the third power.

Boolean Expression : ((A OR B) AND C)

A	B	C	((A OR B) AND C)
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	F
F	F	F	F

Figure 7. Boolean Expression Input Combinations

Path coverage testing of the boolean expression in Figure 7 requires that only two sets of inputs need to be considered. One set would represent a true value of the boolean expression and the other set would represent a false

value. Normal symbolic execution of this example, would result in six sets of possible input conditions that would remain untested. This means that it is possible for undetected errors to remain for six out of eight sets of input conditions. For more complicated boolean expressions, the potential for undetected errors increases.

Multiple condition coverage testing requires that all eight sets of input conditions from Figure 7 be considered. As a result, the symbolic execution of the boolean expression must accommodate all the possible input combinations. Thus, in the symbolic execution tree for the example, there is an eight-way branching from the node representing the statement containing the boolean expression. In general, the branching factor in the tree for boolean expressions must be the same value as the number of sets of input combinations. By making this extension, symbolic execution can provide multiple condition coverage for boolean expressions.

One aspect of boolean expressions in the Ada language that is not found in other languages is the ability to test the equality of entire data structures. For example, a simple relational expression could be two arrays of the same type being tested for equality. If every item in one array is identical to the item in the corresponding position of the other array, then the simple relational expression

evaluates to true. Otherwise, the expression evaluates to false. Theoretically, the symbolic execution tree for this particular situation should contain branches for every possible combination of one or more array position values being unequal in addition to a branch for all being equal. This same type of statement holds true for whatever data structure is being tested for equality. This combinatorial explosion of branching is also compounded by the extended branching already discussed. For example, suppose a boolean expression contained a mixture of equality tests of data structures and other relational expressions. The branching from this boolean expression would be a product of all the possible branches from each relational expression. Clearly, this theoretical method of branching is very impractical.

A more practical solution to this situation is to simply allow the equality or inequality of the expression to evaluate to the values of true or false. If it is important to the source program that a certain pair of values in the data structures be equal or unequal, this could be detected using other analysis techniques, such as data flow analysis. This solution, in effect, adds no other branching into the symbolic execution of boolean expressions other than what has already been discussed.

Appendix A contains a sample Ada program which has at least one error. This error occurs in a boolean expression

coverage symbolic execution can provide additional branching at arithmetic expressions for overflow and underflow. Should the arithmetic expression contain a division operation, another branch is required in the symbolic execution tree to reflect the possibility of division by zero. All of these branches are in addition to the standard branch for normal execution without an error.

Mathematical functions that are discontinuous or are undefined over a range of possible inputs are also potential avenues for numeric errors. An automated symbolic execution tool would be unable to to distinguish a user defined mathematical function from any other user defined function. In either case, the symbolic execution for the function call would have to follow the normal method of branching according to the I/O pairs generated from the symbolic execution of the function.

Symbolic execution could be specifically tailored to accomodate predefined mathematical functions for a particular programming language. The logarithmic function is undefined for negative numbers. Inverse trigonometric functions are discontinuous at periodic intervals. These are examples of predefined functions for which symbolic execution could be sensitive to a particular language. The developer for a specific symbolic execution tool would need to compile a list of the predefined functions for the target

source language. Branching for the undefined and discontinuous error possibilities could then be built into the symbolic execution tree for these particular function calls.

3.4 Arrays

An array is a data structure that allows a related group of data to be referenced by a common name. Individual data items are stored into and retrieved from an array via an index. Richardson and Clark [3] identify array indexing as one weakness in symbolic execution. If the array is indexed by a variable, then there is no guarantee at run time that the variable is within the bounds of the array. Hence, it becomes necessary to provide an additional branch in the symbolic execution tree each time an array is encountered for the possibility of an index out of range.

In the case of an n-dimensional array, the number of possible input conditions on the array indices is two to the nth power. The input conditions represent whether the array indices are within or outside the constraints for their respective index positions. From a theoretical standpoint, symbolic execution should provide a branch in the tree for each of these possible input conditions. From a practical standpoint, the symbolic execution tree would become very large for even very small and simple programs. An actual implementation of this may benefit from creating only a two-

way branch at each array. One branch represents that all of the indices are within their bounds. The other branch represents the possibility that any combination of one or more indices is out of the bounds on its range. For most programs, having any one array index out of bounds would be a logic error that would require correction. Having only one branch to represent all of the error conditions on the indices does not provice for multiple condition coverage. But, it may end up being a practical method of implementing array handling in symbolic execution.

3.5 Attributes

The Ada language provides a set of constructs called attributes which can be used to determine certain properties of types, objects and subtypes during execution [2]. These attributes are formed by appending an apostrophe and the attribute name onto the end of a variable name. The attributes that are of interest in terms of symbolic execution are SUCC, PRED, and VALUE.

These attributes are actually special functions in the Ada language. SUCC returns the value of the next item of its variable's type. For example, if the variable is an integer with the value 6, then the SUCC attribute would return a 7. PRED is similar to SUCC except that it returns the previous item of the variable's type. The VALUE attribute operates on a string of characters that match one

of the items of the variable's type. When this string is operated on by VALUE, it returns the item of the type the string matches.

The possibilities for errors with the SUCC and PRED attributes occur when the variable contains a value at one of the extremes of its type. The SUCC attribute causes a constraint error when the variable contains the last value of its type. Similarly, the PRED attribute causes a constraint error when the variable contains the first value of its type. These errors are possible when the variables are sybtypes or enumeration types. Once again, the symbolic execution tree will contain an extra branch for the possibility of these error conditions any time the SUCC or PRED attribute occurs.

The VALUE attribute has a high potential for error. If the string does not exactly match one of the items of the variable's type, a constraint error occurs. As with the other attributes discussed, the symbolic execution tree needs to have an extra branch for this possible error condition every time the VALUE attribute occurs.

3.6 Access Types

Access types are more generally referred to as pointers. Access types are defined to either reference a particular type of data structure or to contain a value of null [2]. Assigning a null value to an access type may be a

desirable thing to do in many circumstances (i.e. signifying the end of a linked list or a 'leaf' node in a tree, etc...).

The potential for error in using access types occurs when attempting to store or retrieve data when the access type is null. This may occur in three different ways. When an access type is created, it is assigned a default value of null. An access type may be assigned a value of null within the code of the program. Or, an access type may be deallocated.

If two access types reference the same data structure and one of them is deallocated, both access types are set to null. This is known as a dangling reference [2] and it represents one of the major problems with access types. An attempt to store or retrieve data from an access type that is a dangling reference causes a constraint error. So, symbolic execution must be sensitive to this possibility for a null access type (as well as the other two possible ways access types become null) and provide an extra branch in the tree for each occurance of an access type.

3.7 Undefined Variables

At any point within a program, a variable may be referenced which has not yet been assigned a value. Depending on the type of the variable, a number of possible errors may occur. In the case that the variable is a

subtype, a constraint error may occur. While theoretically symbolic execution should account for the possibilities of these errors occuring, it would not be very practical. Creating extra nodes and branches in the symbolic execution tree every time a variable is referenced requires excessive overhead. Keeping a table of every variable which has been defined and then creating a branch for each one referenced that is not in the table has some flaws. For arrays, there is a problem of keeping track of which positions have been defined. For records, the data items of each record have the same names. For these reasons, it is impractical to implement a strategy for undefined variables into symbolic execution and requires another technique such as data flow analysis.

3.8 Input

Data inputs into a program are another source of potential errors. The Ada construct for reading data is the GET or GET_LINE command [2]. In most situations, the data input into the program is what the program expects. However, at times the input may be of a different type than expected. Or, if subtypes are being used, the data may be of the correct type but it may be outside the bounds of the subtype.

Symbolic execution for the GET or GET_LINE command containing n variables requires a branching factor of three

raised to the nth power. This is due to the three possible input conditions mentioned for each variable. A practical method would be to generate a three-way branch in the symbolic execution tree for GET or GET_LINE commands. One branch represents that the inputs are all of the correct type and within their respective constraints. The next branch represents that at least one variable is outside its range of constraints. The third branch represents that at least one variable received data of the wrong type. This method eliminates the branching explosion described above yet it still addresses the error possibilities presented here.

3.9 Discriminants

A discriminant in the Ada language appears as a parameter in a record declaration [2]. It is used to declare records of slightly differing types. The difference between the records is the value of the discriminant.

The problem posed here for symbolic execution is the fact that it is possible for assignments to be made from one record to another of the same type which have different values for their discriminants. If this happens during execution, a constraint error occurs. Therefore, the symbolic execution tree must provide an additional branch for record assignments when the records are defined using discriminants. This extra branch represents the constraint

error that is raised if the discriminants are of different values.

3.10 Subtypes

Subtypes have been mentioned in several of the subsections of this chapter. This subsection is provided to cover subtypes in general.

Subtypes in the Ada language allow for variables that are defined for only a portion or range of another type [2]. The Ada compiler allows assignments from a variable of the parent type to a variable of the subtype. In some cases the variable of the parent type may contain a value outside the range of the subtype. When this occurs, a constraint error is raised. Symbolic execution can provide an extra branch for the possibility of a constraint error each time a value is assigned to a subtype variable.

3.11 Variant Records

Variant records are record types which contain different variables depending upon the value of a case variable in the record. This situation is similar to the one described for discriminants, however, the error that may occur here will do so for a different set of circumstances. A constraint error will result from attempting to use a variable of the record that does not exist due to the value of the case variable. Therefore, the symbolic execution tree must generate an extra branch for each reference to a

selected component of a record that may vary based on the case variable.

3.12 Remarks

For each new branch, a path condition can be generated which is based on the prior pc and the condition causing the branch. This generated path condition characterizes the set of all initial states for the procedure that will cause the error condition. A valuable extension to this approach would involve determining whether that set of initial states is empty. When it is, then the erroneous path is unreachable, and no test case needs to be generated.

For a symbolic execution system to be able to recognize access types, subtypes, records using discriminants, and variant records, it will need to keep type and range of value information for all of the data structures and variables in the source program. This will require even more memory utilization and execution time on top of the greatly increased size and manipulation time of the symbolic execution tree. However, in order to increase the error detection capability of symbolic execution, this extra price must be paid.

3.13 Summary

This chapter presented several Ada expressions, statement types, and constructs for which symbolic execution does not address. An analysis of each of these items was

conducted to provide a theoretical extension for them under symbolic execution. The theoretical extensions for some of the items were found to be impractical. In these cases, more practical yet less comprehensive extensions to symbolic execution were discussed.

Chapter 4

Symbolic Execution Tree Package

The detailed design of the symbolic execution tree package is presented in this chapter. This package along with ALEX and AYACC [10] will be the three major components of the IOGen system. The symbolic execution tree package requires some minor additions within AYACC and the compiler procedure that drives ALEX and AYACC.

4.1 Overview

The basis upon which the symbolic execution tree package is designed is the CAIS list management environment [1]. The CAIS list management environment contains six packages of procedures and functions for list management. These packages are: CAIS_List_Management, CAIS_List_Item, CAIS_Identifier_Item, CAIS_Integer_Item, CAIS_Float_Item, and CAIS_String_Item. Four of these packages are used in this design. The four are the first, second, fourth, and sixth packages listed. The symbolic execution tree structure is easily represented, built, and manipulated in a list form. The advantage to designing the tree based upon lists is the fact that the power and versatility of the CAIS list management environment is utilized.

The symbolic execution tree package designed in this chapter includes the Ada programming constructs described in Chapter 2 and the extensions for boolean expressions

developed in Chapter 3.

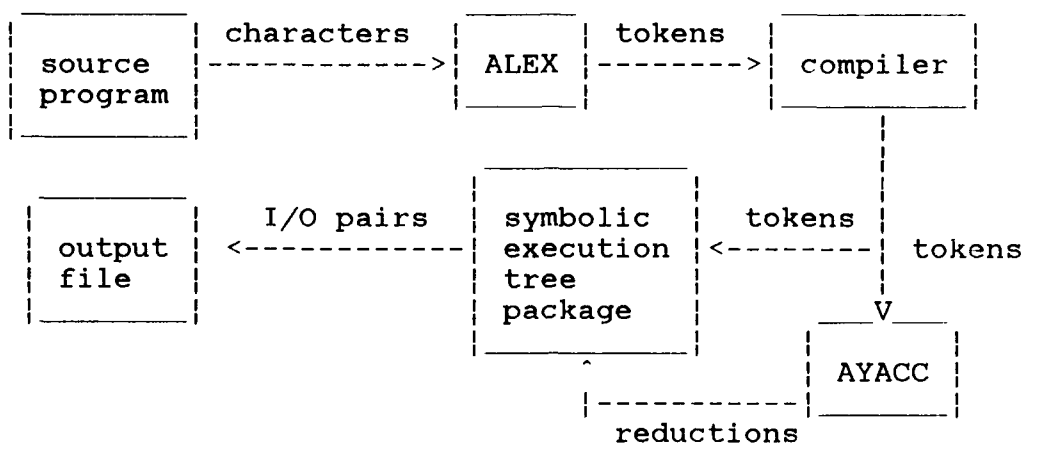


Figure 8. IOGen System Configuration

Figure 8 represents the structure of the major components of this design for IOGen. The compiler is the main driver for the system. It calls the lexical analyzer, ALEX, for tokens which are built of character strings from the source program. Based upon these tokens, the compiler uses the tables in the parser, AYACC, to parse the program. AYACC is an LR(1) parser which parses the program in a bottom-up fashion. The symbolic execution tree package receives the tokens from the compiler as well as procedure calls from the parser as productions are reduced. Once the entire program has been parsed, and the symbolic execution tree is complete, the I/O pairs are contained in the 'leaf' nodes of the tree. The tree is traversed to find the leaf nodes and send the I/O pairs to an output file.

4.2 Design of the Package

The symbolic execution tree package is designed to utilize the CAIS list management environment [1] as well as

---

token_stack : This is a stack of tokens from the source program. These tokens were built by the lexical analyzer and sent to this package by the compiler.

symb_ex_tree : The outermost list of the tree structure.

leaf_list : A list of the leaf nodes in the tree structure at a given point in the symbolic execution.

current_list : The current active list. In most cases this will be the parent node for which new children are being created and inserted into.

new_nodes_table : A table of nodes (lists) being created for insertion into the tree list structure. In many cases, several nodes are being created at the same time (i.e. for boolean expressions).

relations_table : A table of the simple relations that make up a boolean expression. These simple relations are used to generate the multiple input conditions on the boolean expression.

boolean_expression : A string variable that will contain a boolean expression from the source program. This will be used in the boolean evaluation of the multiple input conditions generated from the relations table.

statement_string : A temporary working area for various statement fragments throughout the package.

Figure 9. Data Structures

---

to interact with the compiler and parser. Figure 9 shows the major data structures of the symbolic execution tree package. All of the data structures except the token_stack and the symb_ex_tree are local to each procedure. The token_stack is a global structure that may be used by any of the procedures. The symb_ex_tree represents the entire tree structure and points to the root node.

The general structure of a node in list form is:

A(#_of_sublists,B(),...,N(),pc,oc).

'B(),...,N()' are the sublists or children nodes, 'pc' is the current path condition, and 'oc' is the current set of output conditions.

Figure 10 contains a list of the major procedures in the package. They are only presented here as a consolidated list of the procedures. The detailed algorithms are presented in the following subsections.

```
Procedure Assignment_Statement
Procedure Generate_New_Nodes
Procedure For_Loop
Procedure When_Clause
Procedure Procedure_Calls
```

Figure 10. Package Procedures

The symbolic execution of several Ada constructs was presented in Chapter 2. The constructs were: assignment statements, if_then_else statements, looping constructs,

case statements, and procedure calls. The symbolic execution of these constructs is discussed again in this section with respect to this design. This discussion will center on the processing of each of these Ada statement types from the point they are recognized in the parser until the appropriate nodes have been built and inserted into the tree.

4.2.1 Assignment Statements

The production in the Ada grammar [6] for an assignment statement is:

assignment_statement ::= variable_name := expression ;

When the parser reduces this production, an assignment statement has been recognized. The action associated with this reduction is a call to procedure Assignment_Statement. The algorithm for this procedure is presented in Figure 11.

Figure 12 shows an assignment statement with the before and after versions of the tree list structure. The assignment statement is x := 0 ; . List A contains only the path condition (pc) and output conditions (oc) associated with it prior to generating its sublist (child). List B is the new node created for the assignment statement. List B is assigned the pc from its parent list as the pc does not change in this assignment statement. The oc is ANDed with the result of the assignment statement and becomes the

Procedure Assignment_Statement

- extract the assignment statement tokens off the token_stack and rebuild the statement in the statement_string.
- create a new node using procedure Set_To_Empty_List from the CAIS_List_Management package.
- extract the oc from the current_list using procedure Extract_Value from the CAIS_String_Item package.
- AND the assignment statement in the statement_string with the oc.
- insert the new oc into the new node using procedure Insert from the CAIS_String_Item package.
- extract the pc from the current_list using procedure Extract_Value from the CAIS_String_Item package.
- insert the pc into the new node using procedure Insert from the CAIS_String_Item package.
- replace the pc and oc from the current_list with an empty string using procedure Replace from the CAIS_String_Item package.
- insert this new node into the current_list using procedure Insert from the CAIS_List_Item package.
- insert the value 1 into the current_list as the number of sublists (children) is 1. Use procedure Insert from the CAIS_Number_Item package.
- set the current_list to the new node using procedure Make_This_Item_Current from the CAIS_List_Management package.

Figure 11. Procedure Assignment_Statement

output conditions for list B. List B is inserted into list A and the number of sublists, 1, is inserted into list A as it has only one sublist. From this point the parsing of the

source program and symbolic execution may continue with the next statement in the source program.

```
x := 0 ;
A(pc,oc)    -- before symbolic execution
A(1,B(pc,oc&x=0))    -- after symbolic execution
```

Figure 12. Symbolic Execution of an Assignment Statement

4.2.2 If_Then_Else Statements

The production in the Ada grammar for an if_then_else statement is:

```
if_statement ::= IF condition Then sequence_of_statements
                 (ELSEIF_condition_THEN_sequence_of_statements)
                 /ELSE_sequence_of_statements\ END IF ;  [6].
```

The nonterminal condition is a boolean expression and it is the primary focus of this section and the section on looping constructs. The nonterminal sequence_of_statements is one or more Ada statements. The nonterminal (ELSEIF_...) is an optional construct that allows for an else if part of the statement. This is another means of nesting if statements. The nonterminal /ELSE_...\ is also an optional construct that allows for a final else part of the if statement.

Previously, symbolic execution of boolean expressions has only provided a two-way branching in the symbolic execution tree. From the discussion on boolean expressions in Chapter 3, it is apparent that the branching factor must

accomodate the multiple input conditions that can occur in the boolean expression. In particular, this branching factor was determined to be 2 raised to the nth power for a boolean expression containing n simple relational operators. In some cases, a boolean expression may have several simple relations and the branching factor may get rather large. However, in most cases, the number of simple relational operators will not be more than two or three. So, the branching factor will usually not exceed eight for a given boolean expression. In light of this, modifying symbolic execution to branch according to the multiple input conditions is practical and will provide a higher level of confidence in the correctness of a program.

Prior to recognizing the if_then_else statement, the parser will recognize its boolean expression. While the parser is in the process of reducing tokens and nonterminals to a boolean expression, it will recognize each of its simple relational expressions. At the point that each simple relational expression is reduced, the tokens for that expression are on the top of the token_stack. When the reduction occurs, the action is to place a copy of the tokens for the expression into the relations_table. When the reduction occurs for the entire boolean expression, all of the simple relational expressions are in the relations_table. At this point, all of the tokens that make

Procedure Generate_New_Nodes

- remove the tokens for the boolean expression from the token stack and place them into the boolean_expression string.

- extract the pc and oc from the current_list using procedure Extract_Value from the CAIS_String_Item package.

- generate the multiple input conditions based upon the relations in the relations table.

- for each set of input conditions loop :

  -- create a new node using procedure Set_To_Empty_List from the CAIS_List_Management package.

  -- AND the pc from the parent list with the input conditions for the current loop.

  -- insert the oc into the new node using procedure Insert from the CAIS_String_Item package.

  -- insert the new pc into the new node using procedure Insert from the CAIS_String_Item package.

  -- evaluate the input conditions against the boolean_expression. If the value is true, place a true marker into the new node. Otherwise place a false marker in the new node. Use procedure Insert from the CAIS_String_Item package.

  -- insert the new node into the current_list using procedure Insert from the CAIS_List_Item package.

- insert the number of new nodes into the current_list using procedure Insert from the CAIS_Number_Item package.

Figure 13. Procedure Generate_New_Nodes

up the boolean expression are on the top of the token stack. The action for the reduction of the boolean expression is a call to procedure Generate_New_Nodes. Figure 13 shows the

algorithm for this procedure.

As the parsing continues, the 'then' portion of the if_then_else statement is encountered. The next reduction to take place in the parser that is of interest is for the first statement in the then portion. The sequence of statements that make up the then portion are symbolically executed prior to the parser reducing then to the sequence_of_statement nonterminal. In other words, since the parse proceeds left-to-right while the reductions occur in a bottom-up manner, nested statements must be symbolically executed prior to the symbolic execution of the nesting statement. This implies a rather recursive manner to the symbolic execution when one or more statements is nested within another statement. In the case of the if_then_else statement, the boolean expression is reduced and can be processed before the nested statements of the then and else parts are encountered. So, the new nodes created in procedure Generate_New_Nodes will exist when the point is reached to begin symbolically executing the nested statements.

The symbolic execution of these nested statements will result in nodes with incomplete pc's and oc's. This is due to the fact that these statements are symbolically executed without knowledge of which node would become its parent. Upon the reduction to the nonterminal sequence_of_statements

for the 'then' part, the sequence will be in a partially completed list structure. However, this list structure will not have a parent at that moment. For the case that this list represents the then portion of an if_then_else statement, a copy of this list can be placed into each node that has a true marker from procedure Generate_New_Nodes. Copies of the list can be created by calling the procedure Copy_List from the CAIS_List_Management package. This procedure will have to be called for each copy that is needed. The only remaining step to perform is to AND the pc and oc of the node receiving the list with every partial pc and oc in the list structure.

This same recursive process must occur for the 'else' portion of the if_then_else statement (if one exists). A copy of the new list structure must be inserted into each of the nodes containing a false marker. Then the pc and oc for every node in the list structure must be updated as described above. The true and false markers can be removed once the list is inserted into the node.

In the event that the if_then_else statement has an 'elseif' part, this represents a nesting of another if statement in the else part of the outer statement. Again, the same process applies to this nested if_then_else statement as before. The only difference is that each node that evaluated to false at the outer level obtains as

children a complete set of nodes generated from the symbolic execution of the nested boolean expression within the 'elseif' part. This process repeats itself recursively each time another 'elseif' construct is encountered.

Once the final 'else' part is symbolically executed and all the lists are inserted into their parent lists (with pc's and oc's updated), the parser recognizes the outer if_then_else statement and performs the reduction. From this point, the parsing may continue on to the next statement in the source program. Symbolic execution will proceed for each leaf node in the list structure.

Figure 14 shows an example of an if_then_else statement with a somewhat complex boolean expression. Prior to symbolically executing the statement, the parent is simply the list A. The boolean expression contains three simple relational expressions U, V, and W. Accordingly, these are placed in the relations_table. From these three expressions, there are eight sets of input combinations for the boolean expression. All of these input combinations are evaluated and new nodes are created for each set. The list structure is shown for the completed symbolic execution for the entire if_then_else statement. Lists B thru I are sublists of list A but are shown outside of list A for clarity.

```
If ((U AND V) OR W) Then  -- if_then_else statement
  x := 0;
Else
  x := 1;
End If;


A(pc,oc)  -- list of parent before symbolic execution


A(8,B(_),C(_),D(_),E(_),F(_),G(_),H(_),I(_)) -- after
B(1,J(pc&U&V&W,oc&x=0)
C(1,K(pc&U&V&!W,oc&x=0)
D(1,L(pc&U&!V&W,oc&x=0)
E(1,M(pc&U&!V&!W,oc&x=1)
F(1,N(pc&!U&V&W,oc&x=0)
G(1,O(pc&!U&V&!W,oc&x=1)
H(1,P(pc&!U&!V&W,oc&x=0)
I(1,Q(pc&!U&!V&!W,oc&x=1)
```

Figure 14. Symbolic Execution of an If_Then_Else Statement

4.2.3 Looping Constructs

The Ada looping constructs introduced in Chapter 2 were: for, while, and exit_when constructs. The while and exit_when constructs both involve boolean expressions upon which the decision to remain in the loop or leave the loop lies. The for loop does not involve a boolean expression but rather the number of times the loop iterates is determined prior to entering the loop.

The production in the Ada grammar for the for loop specification is:

iteration_scheme ::= FOR loop_parameter_specification [6].

When this production is recognized and reduced, the action

taken is to call procedure For_Loop. Figure 15 shows the algorithm for this procedure.

Procedure For_Loop

- extract the loop parameter specification tokens from the token_stack and place them in the statement_string.
- create two new nodes using procedure Set_To_Empty_List from the CAIS_List_Management package.
- extract the pc and oc from the current_list using procedure Extract_Value from the CAIS_String_Item package.
- insert the oc into both of the new nodes using procedure Insert from the CAIS_String_Item package.
- AND the pc with 'loop_variable = first_loop_value'
- insert this new pc into one of the new nodes using procedure Insert from the CAIS_String_Item package.
- AND the old pc with 'loop_variable not in range'
- insert this new pc into the other new node using procedure Insert from the CAIS_String_Item package.
- insert both new nodes into the current_list using procedure Insert from the CAIS_List_Item package.
- insert a 2 into the current_list using procedure Insert from the CAIS_Number_Item package.

Figure 15. Procedure For_Loop

Procedure For_Loop creates a two-way branch in the symbolic execution for a for loop as described in Chapter 2. Symbolic execution continues for the node whose branch enters the loop. The body of the loop is a sequence of statements just as the 'then' part of the if_then_else

statement contained a sequence of statements. This sequence is symbolically executed and the resulting list becomes a sublist of the node entering the loop. Upon encountering the end of the loop, the grammar production for a loop_statement is reduced. When this occurs, the symbolic execution for both branches with the next statement following the loop.

Figure 16 shows a for loop with its corresponding before and after tree list structures.

```
For i := 1 to n loop
  x := x+1;
End Loop;

A(pc,oc)    -- before

A(2,B(_),C(_))    -- after
B(1,D(pc&i=1,oc&x=x+1),pc&i=1,oc)
C(pc&i<>1..n,oc)
```

Figure 16. Symbolic Execution of a For Loop

Note that the branch node entering the loop, B, also contains the node for the assignment statement within the loop. This represents that the entire loop has been symbolically executed and the process may continue with the next statement in the source program.

The production that indicates a while loop in the Ada grammar is:

iteration_scheme ::= WHILE condition [6].

Just as in the if_then_else statement, the nonterminal condition is a boolean expression. When the boolean expression is recognized in the parser, the procedure Generate_New_Nodes is called as described earlier.

Similar to the then part of the if_then_else statement, the sequence of statements in the while loop must be symbolically executed prior to the parser recognizing the while loop body. The same nested process as described before must be undertaken. The list structure representing one iteration through the loop body must be copied and inserted into each node that evaluated to true. Again, all pc's and oc's in the list must be updated as described earlier. Once this is complete, the parser reduces the loop construct and symbolic execution may continue for every leaf node in the tree list structure.

```
While (U AND V) Loop
  x := x+y;
End Loop ;


A(pc,oc)    -- before


A(4,B(_),C(_),D(_),E(_))    -- after
B(1,F(pc&U&V,oc&x=x+y),pc&U&V,oc)
C(pc&U&!V,oc)
D(pc&!U&V,oc)
E(pc&!U&!V,oc)
```

Figure 17. Symbolic Execution of a While Loop

Again, note that the branch node, B, that goes into the loop also contains a node, F, for the assignment statement in the loop. Furthermore, the boolean expression contains two simple relational expressions which require the four-way branching of the loop.

The exit_when construct is similar to the while loop except that the exit condition (boolean expression) does not necessarily occur at the beginning of the loop. Following the process for the exit_when construct presented in Chapter 2, the symbolic execution of the loop up to the exit_when statement will proceed as if a loop has not been entered. This is due to the fact that loops are only symbolically executed once and there has been no branching due to the loop structure to this point. Once the exit_when statement is encountered, it is handled exactly as the while loop except that the nodes whose boolean expressions evaluated to false enter the loop. Those nodes whose boolean expressions evaluated to true skip the rest of the loop. As with all other constructs, symbolic execution continues for all leaf nodes following the end of the loop.

4.2.4 Case Statement

The production in the Ada grammar for the case statement is:

```
case_statement ::= CASE expression IS
    case_statement_alternative(case_statement_alternative)
```

END CASE ; [6].

The nonterminal expression represents some type of variable such as a simple variable, array position, record variable etc... The nonterminal case_...(...) represents a sequence of when clauses for the case statement.

The first reduction of interest in the parser is the reduction to the nonterminal expression. When this reduction occurs, the variable is taken off the token_stack and placed in the statement_string. This is used to update the pc's later in the process.

The next reductions to occur are all involved in the statements that comprise the body of the first when clause. Again, as in the if_then_else statement, a recursive level of symbolic execution must occur to process the sequence of statements of the when clause. This repeats for each when clause encountered in the case statement. Once the sequence of statements is processed and the list structure is built, the reduction for the individual when clause occurs. At this point the action from the parser is to call procedure When_Clause.

Figure 18 shows the algorithm for procedure When_Clause. This procedure completes the symbolic execution tree for one of the options in the case statement. It will be called once for each when clause reduced. Upon completion of the case statement, symbolic execution

Procedure When_Clause

- extract the pc and oc from the current_list using procedure Extract_Value from the CAIS_String_Item package.
- create a new node for this when clause using procedure Set_To_Empty_List from the CAIS_List_Management package.
- insert the oc into the new node using procedure Insert from the CAIS_String_Item package.
- extract the case option value from the token_stackl
- append the variable in the statement_string with an equal sign and the case option value. If the value is 'others' this will become a sequence of variable <> all other option values.
- AND this input condition with the pc from the parent node.
- insert this new pc into the new node using procedure Insert from the CAIS_String_Item package.
- insert the list structure for the sequence of statements into the new node using procedure Insert from the CAIS_List_Item package.
- update the pc's and oc's in the inserted list as described earlier.

Figure 18. Procedure When_Clause

continues with the next statement for each leaf node in the tree list structure.

Figure 19 shows a case statement with the list structure from both before and after symbolic execution. Again, note that the list structures for nodes B thru D are shown outside list A for clarity.

```
Case x Is

  When 0 => flag := false;
  When 1 => flag := true;
  When others => flag := true;

End Case ;


A(pc,oc)   -- before


A(3,B(_),C(_),D(_))   -- after
B(pc&x=0,oc&flag=false)
C(pc&x=1,oc&flag=true)
D(pc&x<>0&x<>1,oc&flag=true)
```

Figure 19. Symbolic Execution of a Case Statement

4.2.5 Procedure Calls

The Ada grammar production for a procedure call statement is:

entry_call_statement ::= entry_name/actual_parameter_part\ ;

The nonterminal entry_name/...\ contains the procedure name and a parameter if one exists. When the reduction for the entry_call_statement nonterminal occurs, the procedure Procedure_Call is called. This procedure is shown in Figure 20.

The process of creating an n-way branch in the symbolic execution tree for the n I/O pairs from a procedure call was discussed in Chapter 2. This results in a considerable amount of time saved by not having to execute the procedure each time a call to it is encountered. Rather, it is

Procedure Procedure_Call

- extract the pc and oc from the current_list using procedure Extract_Value from the CAIS_String_Item package.
- for each I/O pair from the symbolic execution of the named procedure loop :
  - -- create a new node using procedure Set_To_Empty_List from the CAIS_List_Management package.
  - -- AND the oc from the parent node with the oc from the I/O pair.
  - -- insert this new oc into the new node using procedure Insert from the CAIS_String_Item package.
  - -- AND the pc from the parent node with the pc from the I/O pair.
  - -- insert this new pc into the new node using procedure Insert from the CAIS_String_Item package.
  - -- insert the new node into the current_list (parent) using procedure Insert from the CAIS_List_Item package.
- insert the number of new nodes into the parent node using procedure Insert from the CAIS_Number_Item package.

Figure 20. Procedure Procedure_Call

symbolically executed once, and the I/O pairs are saved to an output file. When the procedure is called in the source program, the list of I/O pairs is all that is needed to expand the tree for the procedure call. Upon completing symbolic execution of the procedure call, the execution may continue with the next statement in the source program as

before.

4.2.6 Design Wrap-Up

At the end of each subsection a statement is made about how the symbolic execution continues after the statement was completes. In particular, the symbolic execution continues to the next statement for each leaf node in the tree list structure. The leaf_list from the data structures in Figure 9 contains a list of the leaves in the tree list structure.

Upon completing the symbolic execution for a given statememt type, the leaves in the list structure are placed in the leaf list. As the symbolic execution for the next statement completes, a copy of the subtree list is made for each node in the leaf_list. This can be accomplished using procedure Copy_List from the CAIS_List_Management package. Each copy must be traversed to update the pc and oc as described earlier for nested statements. As each copy is being traversed, its leaves are saved in a new leaf_list for the next statement symbolically executed. Once all the copies have been traversed, the execution proceeds to the next source program statement.

Finally, at the end of the source program, the pc's and oc's of the leaf nodes make up the I/O pairs for the program. These pairs are saved in an output file for further use or to be printed out as results.

4.3 Summary

This chapter has presented the detailed design of the symbolic execution tree package for the IOGen system. The major data structures and procedures for this package have been described in detail as well as being presented in the figures. Symbolic execution for all of the Ada constructs discussed in Chapter 2 was covered along with the extension for handling boolean expressions.

Chapter 5

Case Study

The sample program in appendix A was developed to provide a case study in the improved error detection capability of IOGen. Program triangle takes as input the lengths of the three sides of a triangle in descending order. It then determines the classification of the triangle. In this case, the triangle classifications are: equilateral, isosceles, acute, obtuse, and right. If the lengths provided to the program are out of order or do not represent an actual triangle, then the program returns an invalid indication.

The triangle program has one error purposely inserted to test the two methods of symbolic execution. The two methods are embodied in the processing of the IOGen system before and after the design of Chapter 4. The error in the program occurs in the third if statement. The statement should read : If ((A = B) AND (B = C)) Then...
An extraneous greater than side was inserted in the right simple relational expression. The result is that in some cases an isosceles triangle is incorrectly classified as an equilateral triangle.

From the path condition coverage symbolic execution of the program, six I/O pairs are generated. These I/O pairs are presented in the appendix along with a set of test

derived from these pairs. One test set and its resulting output from the program is shown for each I/O pair. Clearly, it can be seen that it is possible for a rather innocent appearing error to remain undetected. In this case the error is not revealed.

The I/O pairs from the extended version of IOGen are presented next in the appendix. Upon creating branches for the multiple input conditions on the boolean expressions, there are 22 I/O pairs generated for the same program. One interesting item to note that has already been mentioned is that half of the I/O pairs contain unreachable paths. This means that input conditions for these eleven pairs are impossible to satisfy.

From the remaining eleven I/O pairs that are possible to execute, a set of test cases was developed. These test cases are shown following the I/O pairs in the appendix. The second test set reveals the error in the program. For the inputs of 4, 4, and 3 (which represents an isosceles triangle) the program indicates that the triangle is equilateral. This is exactly the error described earlier. So, by symbolically executing the program based on the design presented in Chapter 4, the error is now detected.

This case study has illustrated the extra benefit afforded by extending symbolic execution to cover the multiple input conditions on boolean expressions. This

chapter has outlined the execution of the sample program shown in Appendix A for two methods. This case study has also raised one issue that warrants further analysis and research in relation to boolean expressions. This issue pertains to the generation of impossible input conditions for a large percentage of the I/O pairs. This is dicsussed in Chapter 6 in more detail.

Chapter 6

Current and Future Research for IOGen

Other topics of research in relation to IOGen presently being conducted primarily focus on exception handling and tasking. Exceptions can be subdivided into two categories. One category is implicit exceptions. These are the exceptions raised by the Ada language environment when a run time error occurs in the execution of an Ada program. Some of the discussion in Chapter 3 of this thesis touches on this topic. The other category is explicit exceptions. These are exceptions defined within the body of the source program. These two categories of exception handling are the basis of research for another thesis concerning IOGen.

The notion of applying symbolic execution to the Ada tasking capability is the focus of another area of research. Symbolic execution is a static analysis technique. Tasking, on the other hand, introduces parallel activity for a program to perform. Needless to say, it remains a very difficult chore to extend the theoretical and practical basis of symbolic execution to accomodate the temporal information in tasking.

One area of future research that occurs as a direct result of this thesis pertains to simplifying the path conditions for many of the nodes in the tree. In generating nodes for all of the multiple input conditions to boolean

expressions, several of the resulting paths are logically impossible to execute. From the sample program and its I/O pairs in Appendix A, it can be seen that quite a number of impossible paths are generated. If a mechanism were designed into IOGen to simplify and identify contradictory path conditions prior to generating nodes for them, these branches in the symbolic execution tree could be pruned out of the structure. This would result in the savings of a considerable amount of memory and execution time. An extension that would save time and memory would be a worthwhile extension.

This chapter has presented a brief summary of two current areas of research related to the IOGen system. Also, one area of future research has been suggested. This future research could result in the streamlining of the IOGen system to make it much more time and memory efficient.

Chapter 7

Conclusion

This thesis addresses extensions to symbolic execution [8] for the Ada programming language. The detailed design for a new symbolic execution tree package is presented in Chapter 4 and it includes the simple Ada statement types of : assignment statements, if_then_else statements, looping constructs, case statements, and procedure calls. The extensions described for accomodating multiple input conditions on boolean expressions are also built into the design. The symbolic execution tree package utilizes the CAIS list management environment and the interactions between the package and the environment were discussed. The interaction between the package and the other two components of the system, ALEX and AYACC, was discussed.

A case study was presented to illustrate the improved capability of the IOGen system with the design from this thesis. The case study introduced an Ada program with an error in it. This error was not detected by the method IOGen currently utilizes. Under the new design, the error was detected.

Finally, two areas of current research related to the same topic were briefly mentioned. Also, one suggestion for possible future research was described.

This thesis provides a general framework for beginning

the enhancement of the IOGen system from one that only provides assistance in path coverage testing to one that assists with multiple condition coverage testing. This system is still in its early stages of development. Upon completion of several of the current and future research issues related to IOGen, it should prove to be an invaluable aid to the testing and validation of Ada programs.

References

[1] Ada Joint Program Office. Military Standard Common Ada Programming Support Environment (APSE) Interface Set (CAIS). Department of Defense. pp. 419-490, 1986.

[2] G. Booch, Software Engineering With Ada. Menlo Park, CA: Benjamin/Cummings, 1983.

[3] D. Clark and L. Richardson, "Applications of symbolic evaluation," in Journal of Systems and Software, vol. 5, no. 1, pp. 15-35, Jan. 1985.

[4] R. Dunn, Software Defect Removal. New York: McGraw-Hill, 1984.

[5] J. L. Facemire and T. E. Lindquist, "Using an Ada-based abstract machine description of CAIS to generate validation tests," in Washington Ada Symposium, Washington D. C., 1985.

[6] G. Fisher, "A LALR(1) grammar for ANSI Ada," in ACM Ada Letters, vol. III, no. 4, pp. 37-50, Jan./Feb. 1984.

[7] S. L. Hantler and J. C. King, "An introduction to proving the correctness of programs," in ACM Computing Surveys, vol. 8, no. 3, pp. 331-353, Sept. 1976.

[8] J. R. Jenkins, "Automated generation of input/output pairs for the CAIS validation test suite," MS Thesis, Department of Computer Science, Arizona State University, Tempe, AZ. May. 1986.

[9] J. R. Jenkins and T. E. Lindquist, "Test-case generation with IOGEN," in IEEE Software, pp. 72-79, Jan. 1988.

[10] Jian, "ALEX, AYACC," MS Thesis, Department of Computer Science, Arizona State University, Tempe, AZ. May. 1987.

[11] T. E. Lindquist, "Research in Ada interface validation and the CAIS operational definition," Research paper, Department of Computer Science, Arizona State University, Tempe, AZ. Sept. 1986.

[12] A. D. McGettrick, Program Verification Using Ada. New York: Cambridge University Press, 1982.

Appendix A

Sample Ada Program and I/O Pairs

```
Procedure Triangle Is

  A,B,C,D : Integer;

Begin

  Get(A);
  Get(B);
  Get(C);
  If ((A >= B) AND (B >= C) AND (A < B+C)) Then
    If ((A = B) OR (B = C)) Then
      If ((A = B) AND (B >= C)) Then    -- error occurs here
        Put("Equilateral");
      Else
        Put("Isosceles");
      End If;
    Else
      A := A*A;
      B := B*B;
      C := C*C;
      D := B+C;
      If (A /= D) Then
        If (A < D) Then
          Put("Acute");
        Else
          Put("Obtuse");
        End If;
      Else
        Put('Right");
      End If;
    End If;
  Else
    Put("Invalid");
  End If;

End Triangle;
```

I/O Pairs from Path Coverage

```
I1: (A >= B) & (B >= C) & (A < B+C) & ((A = B) OR (B = C))
    & (A = B) & (B >= C)
O1: Equilateral

I2: (A >= B) & (B >= C) & (A < B+C) & ((A = B) OR (B = C))
    & !((A = B) & (B >= C))
O2: Isosceles

I3: (A >= B) & (B >= C) & (A < B+C) & !((A = B) OR (B = C))
    & (A*A /= B*B+C*C) & (A*A < B*B+C*C)
O3: Acute

I4: (A >= B) & (B >= C) & (A < B+C) & !((A = B) OR (B = C))
    & (A*A /= B*B+C*C) & !(A*A < B*B+C*C)
O4: Obtuse

I5: (A >= B) & (B >= C) & (A < B+C) & !((A = B) OR (B = C))
    & !(A*A /= B*B+C*C)
O5: Right

I6: !((A >= B) & (B >= C) & (A < B+C))
O6: Invalid
```

Test Cases

Test Set	Actual Output
1. A = 3, B = 3, C = 3	Equilateral
2. A = 4, B = 4, C = 2	Isosceles
3. A = 7, B = 6, C = 5	Acute
4. A = 4, B = 3, C = 2	Obtuse
5. A = 5, B = 4, C = 3	Right
6. A = 9, B = 2, C = 2	Invalid

The error is not detected

I/O Pairs from Multiple Condition Coverage

```
I1:  (A >= B) & (B >= C) & (A < B+C) & (A = B) & (B = C) &
     (A = B) & (B >= C)
O1:  Equilateral

I2:  (A >= B) & (B >= C) & (A < B+C) & (A = B) & (B = C) &
     (A = B) & !(B >= C)    -- Impossible
O2:  Isosceles

I3:  (A >= B) & (B >= C) & (A < B+C) & (A = B) & (B = C) &
     !(A = B) & (B >= C)    -- Impossible
O3:  Isosceles

I4:  (A >= B) & (B >= C) & (A < B+C) & (A = B) & (B = C) &
     !(A = B) & !(B >= C)   -- Impossible
O4:  Isosceles

I5:  (A >= B) & (B >= C) & (A < B+C) & (A = B) & !(B = C) &
     (A = B) & (B >= C)
O5:  Equilateral

I6:  (A >= B) & (B >= C) & (A < B+C) & (A = B) & !(B = C) &
     (A = B) & !(B >= C)    -- Impossible
O6:  Isosceles

I7:  (A >= B) & (B >= C) & (A < B+C) & (A = B) & !(B = C) &
     !(A = B) & (B >= C)    -- Impossible
O7:  Isosceles

I8:  (A >= B) & (B >= C) & (A < B+C) & (A = B) & !(B = C) &
     !(A = B) & !(B >= C)   -- Impossible
O8:  Isosceles

I9:  (A >= B) & (B >= C) & (A < B+C) & !(A = B) & (B = C) &
     (A = B) & (B >= C)     -- Impossible
O9:  Equilateral

I10: (A >= B) & (B >= C) & (A < B+C) & !(A = B) & (B = C) &
     (A = B) & !(B >= C)    -- Impossible
O10: Isosceles

I11: (A >= B) & (B >= C) & (A < B+C) & !(A = B) & (B = C) &
     !(A = B) & (B >= C)
O11: Isosceles
```

```
I12: (A >= B) & (B >= C) & (A < B+C) & !(A = B) & (B = C) &
     !(A = B) & !(B >= C)  -- Impossible
O12: Isosceles

I13: (A >= B) & (B >= C) & (A < B+C) & !(A = B) & !(B = C) &
     (A*A /= B*B+C*C) & (A*A < B*B+C*C)
O13: Acute

I14: (A >= B) & (B >= C) & (A < B+C) & !(A = B) & !(B = C) &
     (A*A /= B*B+C*C) & !(A*A < B*B+C*C)
O14: Obtuse

I15: (A >= B) & (B >= C) & (A < B+C) & !(A = B) & !(B = C) &
     !(A*A /= B*B+C*C) & !(A*A < B*B+C*C)
O15: Right

I16: (A >= B) & (B >= C) & !(A < B+C)
O16: Invalid

I17: (A >= B) & !(B >= C) & (A < B+C)
O17: Invalid

I18: (A >= B) & !(B >= C) & !(A < B+C)
O18: Invalid

I19: !(A >= B) & (B >= C) & (A < B+C)
O19: Invalid

I20: !(A >= B) & (B >= C) & !(A < B+C)    -- Impossible
O20: Invalid

I21: !(A >= B) & !(B >= C) & (A < B+C)
O21: Invalid

I22: !(A >= B) & !(B >= C) & !(A < B+C)  -- Impossible
O22: Invalid
```

Test Cases

Test Sets	Actual Output
1. A = 3, B = 3, C = 3	Equilateral
2. A = 4, B = 4, C = 3	Equilateral - NO!
3. A = 4, B = 3, C = 3	Isosceles
4. A = 7, B = 6, C = 5	Acute
5. A = 4, B = 3, C = 2	Obtuse
6. A = 5, B = 4, C = 3	Right
7. A = 4, B = 3, C = 2	Invalid
8. A = 6, B = 4, C = 5	Invalid
9. A = 4, B = 2, C = 3	Invalid
10. A = 2, B = 4, C = 3	Invalid
11. A = 2, B = 3, C = 4	Invalid

The error is detected.